Insights from expert software design practice

Insights from Expert Software Design Practice

Marian Petre
Centre for Research in Computing
The Open University
Milton Keynes, MK7 6AA, UK
+44 1908 65 33 73
m.petre@open.ac.uk

ABSTRACT
Software is a designed artifact. In other design disciplines, such as architecture, there is a well-established tradition of design studies which inform not only the discipline itself but also tool design, processes, and collaborative work. The 'challenge' of this paper is to consider software from such a 'design studies' perspective. This paper will present a series of observations from empirical studies of expert software designers, and will draw on examples from actual professional practice. It will consider what experts' mental imagery, software visualisations, and sketches suggest about software design thinking. It will also discuss some of the deliberate practices experts use to promote innovation. Finally, it will open discussion on the tensions between observed software design practices and received methodology in software engineering.

Categories and Subject Descriptors
D.2.10 [**Software**]: Design – *methodologies, representations.*

General Terms
Design, Experimentation, Human Factors

Keywords
Expertise, design, empirical studies, software development processes

1. INTRODUCTION
Software design can be difficult. The problems are often "wicked" [Rittel and Webber, 31]: too big, too ill-defined, too complex for easy comprehension and solution. Sometimes the problems are only fully understood after they are solved. Solving such problems is rarely a matter of 'brute force' or routine. So how do expert software designers solve them?

Fundamentally, software engineering is about thinking. Richard Hamming wrote that: *"The purpose of computing is insight, not numbers"* [15]. The constraints to design and innovation within the discipline are not physical, but human: software is constrained primarily by our ability to invent, by what software engineers have managed to think about so far, and how they go about it. Things like algorithms, programming languages, analytic engines, and software solutions are all thought products. Reasoning is at the heart of expertise in software design. How experts reason about problems and design, and how they use representations and tools to help them reason about bigger and more complex things, is interesting.

Researchers have recognised that many software problems are 'too big for the head', and that only exceptional "super-designers" can reason across the full breadth and depth of such massive problems in order to consider consequences and implications of design decisions [Curtis et al., 9]. As a result, expertise is the crucial commodity in software development today, with individual developers differing in productivity by 10 to 30 times [Boehm, 5].

The study of expert design presents particular challenges; expert design is complex in terms of context, task, and time. Design occurs within organisations, teams, and disciplines which shape design processes via social structures and interactions, conventions, and practices (cf. [Bucciarelli, 6]). Software projects may span years, and some aspects of expertise may be occasional phenomena, appearing irregularly. Experts are by definition a minority, and issues of intellectual property and workload limit their accessibility. As a consequence, effective study requires an approach that is sensitive to context and can make the most from small numbers, restricted access, and occasional phenomena. Lawson [19] argued that "…to get good data on [expert design] we need to study not just the actions, graphical outputs and finished designs of these designers but also the conversations they have with each other and their clients during their normal working practice." (p. 38)

Research has shown that experts differ from others not just in the amount they produce, but in how they produce it: they know more, have more effectively organized conceptual representations, solve problems using more advanced processes, use the information they have more effectively, are more creative, and are more pragmatically adept due to their application of their experiences (see [Kaplan et al., 16] and [Allwood, 1] for reviews).

ESEC/FSE '09 Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. ISBN: 978-1-60558-001-2.
ACM Digital Library: http://dl.acm.org

My own research has shown that software design experts are distinguished both by the repertoire of reasoning strategies they use, and by their ability to choose the appropriate strategy for the task. They also build custom tools that match their thinking. So, capturing expert strategy – what experts do, how they reason, and what tools they use – could have a significant impact on productivity. It could also enable us to communicate expert strategies to the next generation of software designers.

Design requires elements of creativity as well as methodical practice and domain knowledge. It is often described as the exploration of the domain space, alternating between expansion of the space of possibilities and the pruning of that space based on requirements and design choices, ideally converging on a satisfactory solution [Newell and Simon, 24]. This exploration is shaped by knowledge (the designer's existing information, experience, ideas, and heuristics that is brought to bear on a design problem), goals (the desired outcome), and ideas (specific notions that together define one or more states in the design space) [Simon, 34] It encompasses both 'normal' design (recognition of known design problems and mapping of known solutions onto them, incremental innovation based on known solutions and solution strategies) and 'radical' design (invention of new solutions for unfamiliar problems) [Vincenti, 35].

Software design has much in common with other design disciplines such as architecture, mechanical engineering, industrial design and so on: for example, the design of structures, the management of multiple and often conflicting constraints, and the need to bridge between conceptual models (the idea of what should be built) and physical models (the pragmatics of what can be built in the world).

Software design is also different in significant ways. Its 'thought products' are abstract, complex, and hard to observe (what does an operating system 'look like'?). Yet these products must also interact with the physical world, and software designers must reason not just about software properties, but also about software's behaviour over time – behaviour which is potentially complex. In other design disciplines, such as architecture, there is a well-established tradition of design studies which inform not only the discipline itself but also tool design, processes, and collaborative work. The 'challenge' of this talk is to consider software from such a 'design studies' perspective, that is, to examine the design process through studies of actual design practice, considering how designers explore the design space [Newell and Simon] – and hence to consider design as a cognitive and social process.

The next sections are organized simply: first, a broad overview of the empirical basis for the insights is offered, along with a characterization of the sorts of expert designers and high performing teams that were studied; then 'insights' from that programme of research are discussed in turn, with perspectives from studies of experts' mental imagery, software visualization tools, sketches, and team behavior, culminating with a more general discussion of implications.

2. BASIS IN EMPIRICAL STUDIES

Understanding how expert software designers and high-performing software design teams create and reason about software design, articulating their strategies, and identifying how they support their reasoning with techniques, tools, and representations, has been the focus of the empirical studies on which the 'insights' presented here are based, studies which are part of on a ongoing programme of empirical research spanning more than 20 years and more than 20 companies.

The research programme has included a spectrum of research methods, from long-term, situated, ethnographically-informed observational studies, through targeted observations and interviews, through constrained tasks in which a number of participants engaged in the same specified task, (also called field experiments or quasi-experiments) to controlled experiments – with other variations such as corpus analyses also within the spectrum. Overall, the emphasis has been on the use of *in situ*, qualitative methods aimed at making the most of limited access to expert designers, while intruding as little as possible on authentic, situated practice. This approach occupies a space between case studies and controlled studies such as experiments and quasi-experiments. It fits into the broad category of work which Ball and Ormerod [6] characterised as 'cognitive ethnography'. It views design activity in context, while contributing to the understanding cognition 'in-the-head', hence attending to "the interplay between people-laden contexts and expert cognition" (p. 148). Ball and Ormerod characterise cognitive ethnography as:

1. observationally specific: using small-scale data collection based around representative time slices of situated activity.
2. purposive: focusing on selected issues within existing work practices, and
3. verifiable: in terms of validating observations across observers, data sets and methodologies.

The approach provides a means for identifying patterns across individuals: identifying phenomena for further study, cataloguing behaviours and strategies, identifying key factors, and focusing questions for further study. It is informed by (and triangulates among) various types of inputs, including: direct observation, talk-aloud protocols, interviews, environments and artefacts. This approach both yields useful descriptive accounts and feeds into other methods such as controlled studies by providing a well-founded basis for focusing investigation. Hence, the spectrum of techniques works together: we observe in order to understand, explicate, abstract into theory, and question – and we constrain, hypothesise, and experiment in order to test and refine our descriptions, explanations, and theory.

The research programme has studied a variety of software designers and developers in a variety of contexts, but largely the focus has been on 'generalist' experts: those designers and problem-solvers who, in mastering a discipline, achieve both breadth and depth. These are creative software designers – those who combine technical expertise with creative flair in conceiving and generating novel solutions and innovative software. The experts, from both industry and academia, and from several countries in Europe and North America, share the same general background: all have ten or more years of programming and software engineering experience; all have experience with large-scale, real-world, real-time, data- and computation-intensive problems; and all are acknowledged by their peers as expert. All are proficient with programming languages in more than one paradigm. The coding language used was not of particular interest

in these investigations, but, for the record, a variety of styles was exercised in the examples, using languages including APL, C, C++, Hypercard, Java, common LISP, macro-assembler, Miranda, Prolog, and SQL. Their preferred language was typically C or C++, because of the control it afforded (although the preference did not exclude routine verbal abuse of the language).

The experts worked in high performing teams: effective, creative, intellectual-property-producing teams that tend to produce appropriate products on time, on budget, and running first time. For the most part, the teams were small teams of 3 to 12 members, and all were in companies where the generation of intellectual property (i.e., novel solutions) and the anticipation of new markets characterised the company's commercial success. All were effective, as evidenced by consistency of turnover, completed projects, and design prizes. Most were in large, long-term (1- to 2-year) projects, with software developers generating between 5 and 10,000 lines of code per compile unit, typically around 200 lines per compile unit, with on the order of 3,000 files per major project.

Industries included computer systems, engineering consultancy, professional audio and video, graphics, computer-aided design and manufacturing, games, embedded systems, satellite and aerospace – as well as retail systems, insurance and telecommunications. Often the software was one component of a multi-disciplinary project including computer hardware and other technology.

It is important to note that these experts work in relatively small companies or groups that typically produce their own software rather than working with legacy systems, although there were examples of the latter in the programme overall. For the most part, the software they produce is 'engineering software' rather than, for example, information systems, although products may include massive data handling and database elements. This context was determined pragmatically – by which companies were willing to allow access to their expert software developers. The results presented may not generalise beyond this variety of design and this style of working.

3. MENTAL IMAGERY AND SOFTWARE VISUALISATION

Elicitation of experts' mental imagery – of the ways they envision and manipulate software designs and programmes in their minds – reveals both that the imagery is rich and varied, and that there are strong commonalities among individuals in terms of the properties of their mental imagery (if not in the entirety of their repertoires) [Petre and Blackwell, 29]. Mental imagery is used here to describe any inspectable mental representations, regardless of the sensory modality of the image. Indeed, the imagery the informants described was not just visual but also verbal, auditory, spatial, and tactile.

Unsurprisingly, the experts' mental imagery was rich and varied, including dynamic mental simulations of **abstract machines** (vivid, colourful, 'physical' structures that could run and be manipulated); strongly spatial imagery corresponding to **landscapes** over which awareness could 'fly', with different parts of the solution residing in different regions; strongly spatial, mathematically-oriented imagery of **solution surfaces** used in "prospecting around the equation space"; an imagery of non-visual **'presence'** (of entities and their relationships "known in the dark"); **verbal imagery**, in which parts of the problem were described or 'discussed' mentally; **"text with animation"**; and even **auditory** presentations of solution characteristics, with auditory qualities like loudness or tone reflecting some aspect of the solution, such as level of activity or type of data.

The nature of this mental imagery – and in particular the characteristics common across different forms of mental imagery – offers insights into how experts think about design.

3.1 *Insight:* Experts' mental imagery supports selection of focus, provisionality, and the juxtaposition of multiple views.

Selection of focus: The imagery afforded tremendous control of attention: of what was considered, of the degree of focus and the level of granularity, and of the level of awareness. The distribution and resolution of information in the imagery was not uniform; the experts chose where to put their attention at any given moment, and different regions of the imagery were described as coming in and out of focus. Information outside the focus might be undefined, or unsolved, or soluble, or solved; mainly, it was deemed not important at the moment. This selection of focus supports experts' ability to reason across the full breadth and depth of software designs in order to consider consequences and implications of design decisions.

Provisionality: All of the imagery could accommodate incompleteness and provisionality, which were usually signalled in the imagery in some way, e.g., absence, fuzziness, partial shading, distance in a landscape, change of tone. This is consistent with Miller's claim [23] that the vagueness of an image is critical to its utility. Accommodating provisionality means that experts can leave decisions open, and hence explicitly maintain options and alternatives.

Juxtaposition and multiplicity: All of the experts described simultaneous, multiple imagery. Some alternatives existed as different regions, some as overlaid or superimposed images, some as different, unconnected mental planes.

3.2 *Insight:* Mental imagery is often externalized as a way of coordinating models of a design.

Mental imagery used by a software designer in constructing an abstract solution to a design problem can be externalised and adopted by the rest of the team as a focal image. The externalized images are used both to convey the proposed solution – to share ideas – and to co-ordinate subsequent design discussions. They tend to be some form of analogy or metaphor, depicting key structural abstractions. But they can also be 'perspective' images: 'if we look at it like this, from this angle, it fits together like this' – a visualization of priorities, of key information flows or of key entities in relationship. Hence, the image is a conceptual configuration which may or may not have any direct correlation to eventual system configuration.

When externalized images are introduced, they are 'interrogated' in discussion by the team, for example establishing its boundaries with questions about 'how is it different from this'; considering consequences with questions like 'if it's like this, does it mean it

also does that?'; assessing its adequacy with questions about how it solved key problems; and seeking its power with questions about what insights it could offer about particular issues. By interrogating and discussing the image and its implications with the originator, the recipients are establishing a shared semantics, and the originator is co-ordinating with the rest of the team. In the course of the discussion and interrogation, the image might be embellished – or abandoned. Sketching is a typical part of the process of assimilation, embodying the transition from 'mental image' to 'external representation'. The sketches may be various, with more than one sketch per image, but a characteristic of a successful focal image is that the 'mature' sketches of it are useful and meaningful to all members of the group. This fits well with the literature about the importance of good external representations in design reasoning (e.g., [Flor & Hutchins, 12], [Schön, 32], and others). Ko, DeLine and Venolia [17], in a study of the information needs of software developers, found that design questions about intent and rationale were among the most difficult to satisfy. These 'mature' sketches, with their shared interpretation, a shared 'jargon' of key terms and short-hand references relating to it, provide a means of preserving intent and rationale within the team.

It is interesting to note that this co-ordination issue has been taken on board by recent software development methodologies, which often try to address it by creating an immersive environment of discourse and artefacts which is intended to promote regular re-calibration with the other team members and with artefacts of the project. For example, 'contextual design' [Beyer and Holtzblatt, 4] describes 'living inside' displays of the external representations in order to internalise the model (to take it into one's thinking), referring to the displayed artefacts as "public memory and conscience". In another example, 'extreme programming' [Beck, 3] emphasises the importance of metaphor, requiring the whole team to subscribe to a metaphor in order to establish that they are all working on the same thing. In that case, the metaphor is carried into the code, for example through naming.

3.3 *Insight:* The software visualizations experts create for themselves are specialised.

The software visualizations and visualization tools which experts build to support their own design activities tend to be designed for a specific context, rather than generic [Petre, 28]. In one expert's characterisation of what distinguished his team's own tool from other packages they had tried: "the home-built tool is closer to the domain and contains domain knowledge". Software developers talk about software visualization with respect to three major activities: comprehension (particularly comprehension of inherited code), debugging, and design reasoning. The visualization tools for design reasoning (the only ones considered here) appeared to fall into two categories, corresponding to the distinction the experts made between 'debugging the software' (reasoning about the software artifact – program visualization) and 'debugging the application' (reasoning about the design, about what is intended – conceptual design visualization).

Low-level or program visualizations tend to be used to debug the software artefact. They pre-suppose that the expert's understanding of the artefact is correct, and they examine the artefact in order to investigate its behaviour. They typically represent the interpreted or implemented design, showing key entities, relationships and structures through different levels of abstraction, allowing the user to examine and manipulate values, for example altering a value of one variable or output from one process while monitoring others and hence identifying the connective relationship between different parts. It appears that these visualizations reflect some aspects of what the imagery presents, but they do not 'look like' what the engineers 'see' in their minds. There are a number of such tools, especially ones that highlight aspects of circuits or code (e.g., signal flows, variables) or tools for data visualization, as well as visualizations that represent aspects of complexity or usage patterns. In effect, they visualize things engineers need to take into account in their reasoning, or things they need in order to form correct mental models, rather than depicting particular mental images.

Conceptual or design visualizations tend to be used to debug the concept or process – to reason about the design. Conceptual visualisations appear to be closer to what engineers 'see' in their minds. (Indeed, examples are often described by the developers as depictions of personal mental imagery.) They often bear strong resemblance to mathematical visualizations or illustrations, for example showing surfaces that relate to solution spaces.

The distinctions between program visualization and conceptual design visualization in the self-built tools are important. Program visualization contributes to the mental imagery rather than reflecting it. Conceptual visualization appears to offer a more direct relationship between the mental imagery and the software visualization. More work is needed on the interaction between the two – to what extent does understanding conceptual visualization contribute to solving problems in the domain of program visualization?

It is important to remember that there are differences between reasoning about conceptual design and reasoning about artefacts. Conceptual design is a divergent thinking problem, in the early stages at least, which requires creativity and readiness to think 'outside the box'. Schön [32] talks about a design as a 'holding environment' for a set of ideas. The importance of fluidity, selectivity, and abstract structure are emphasised by both the experts' own mental imagery and by their stated requirements for visualization tools.

Visualising concepts: There are few visualizations yet to support conceptual design, rather than just re-present the code, performance or data flow. This highlights the need to make available information that is not typically contained in the source code: information about the originators' intentions and models of the software. This implies that the visualizations (and the tools that drive them) must embody more knowledge of the application domain. Experts want to see software visualized in context – not just what the code does, but what it means. Automatic generation from code is inherently unlikely to produce conceptual visualizations because the code does not contain information about intentions and principles.

Domain knowledge: The utility of visualization lies not in mere re-presentation of data, but in an appropriate and meaningful distillation and abstraction of the data in order to provide access to desired information about the software. That is, it is no good translating massive source code into an equally massive visualization; what is required is views on the artefact that disclose significant patterns within it. Minimising cognitive load by reducing the amount of information handled by the user and maximising the information pertinent to the user requires that the

visualization be tailored to the user's task and goals. This, in turn, requires some knowledge of the domain. Because generic tools do not contain domain knowledge, they cannot depict what the software developers actually reason about when they reason about design.

The big issues that face software visualization – particularly with respect to design – relate to matching visualizations to human needs. It is arguable that, currently, what is visualized is what *can be* visualized, not necessarily what *needs to be* visualized. The big technical challenges lie in developing the analysis and selection techniques needed to tailor visualizations to support human cognition. Software developers seek facilities that contribute to insight, e.g., useful abstractions, ready juxtapositions, information about otherwise obscure transformations, informed selection of key information, etc. – and they need those facilities to be set in context, to be informed by domain knowledge. Tools that simply re-present available information (e.g., simplistic diagram generation from program text) do not provide insight.

4. SKETCHES

Conceptual-level reasoning is reflected in the sketches and other informal representations experts make when exploring early design ideas. Notes and sketches allow designers to capture ideas early in the conceptual design process when the ideas are perhaps incomplete and fleeting – these informal representations have a role in capturing, generating, and evaluating design ideas. Lansdowne [18] writes that "…sketching is needed not simply to illustrate completed ideas to others…its main purpose is to assist designers in eliciting, developing and evaluating the design ideas themselves." (p. 1) He reports that good designers are better at externalizing ideas than less able ones, and that they do it earlier in the design process. Cross [8] calls sketching an "intelligence amplifier" and enumerates how sketching helps design thinking: enabling designers to handle different levels of abstraction simultaneously, enabling identification and recall of relevant knowledge, assisting problem structuring through solution attempts, promoting recognition of emergent features and properties (pp. 34-38). Other researchers, too, write about the role of external representations in assisting creativity and cognition.

Conceptual software design is often collaborative. Developers work in face-to-face settings, creating many sketches on paper [Craft and Cairns, 7] or on whiteboards [Damm et al., 10]. It is unusual to see designers get together to discuss a design without making some sort of sketches or notes. If they arrive unprepared to do so, they'll improvise, grabbing whatever means were to hand, for example using marker pens on windows. Not only must designers think about design, but they must also communicate their concepts and coordinate their thinking across the team in order to develop a shared vision.

In this context, sketches can not only capture early ideas, but also potentially communicate them and act as a coordination mechanism to support the design dialogue. Lubars, Potts and Richter [21] conducted a study of the requirements analysis process in 23 organizations, which demonstrated clearly that informal documentation, communication and coordination are all more important during what we now call the conceptual design phase, than conventional notational and analytic methodologies. Luff et al. [22], based on field studies of real-world organizational environments, concluded that paper-based representations had particular advantages for collaboration. The 'tailorability' of paper-based documentation (e.g., its amenability to annotation) and its 'ecological flexibility' (its ability to move around the environment) were key features.

4.1 *Insight:* Designers use, juxtapose, and switch among formalisms deliberately.

Juxtaposition and annotation: Designers use juxtaposition and annotation deliberately and expressively. Experts juxtapose two different representations in order to use the match or mis-match between them to support reasoning and to spot omissions or inconsistencies. They explicitly represent design alternatives in juxtaposition. Annotation, both textual (adding detail, notes, emphasis) and graphical (highlighting, relating) are important and are used dynamically to support dialogues. Crossings out (exclusions, corrections) remain visible in sketches, are referred to explicitly, and are sometimes annotated specifically.

Goldschmidt [14] describes the 'dialectic of sketching': that design is a dialog between the designer and the sketch, in which the externalization plays a key role in cognition, reflection, and creativity. Sketches allow a dialectic between perception of the figural properties in a sketch ('seeing as') and non-figural propositions about the design ('seeing that'), hence "…allows the translation of the particulars of form into generic qualities and generic rules into specific appearances" (p. 139). Schön [33], too, observes the reflective dialog with materials, the externalization of design "talking back" to the designer and providing insight.

Working around formal representations: Software designers use proportionately little free sketching – the majority of their sketches refer to formal representations – yet they consider free sketching to be crucially important in early design. Often, variants of formal representations were interpreted more freely than they might be 'downstream' in the design process. This freedom of expression took the form of using what was directly relevant and useful from the formal representation, disregarding elements that were not, and possibly adding additional elements. Designers might produce incomplete fragments, disregard syntax rules (even commenting that "this wouldn't work like this…", include 'place holder' elements or elements that were not formally part of the notation or component library, add in elements from a different representation, represent alternatives, annotate freely, and so on. Exploiting this freedom allowed the designers to express things that were not addressed or were excluded by the formal representation.

Deliberate changes of formalism: Experts change representation instrumentally and expressively. They make a deliberate change of representation to highlight key points. Further, they deliberately change notation in order to 'escape from the formalism' (and hence the selection, orientation, or simplification) embodied in a given notation and hence to highlight different aspects of a problem or solution.

4.2 *Insight:* Explicit expression of provisionality allows a dialogue with incomplete ideas.

The imprecision, ambiguity and generality of manipulation of free-hand representations is considered by many to be crucial to

design creativity. Goel [13] presents evidence that free-hand sketching "by virtue of being 'dense' and ambiguous – correlates with creative, explorative, ill-structured phases of problem solving and the avoidance of early fixation. Fish and Scrivener [11] argued that creativity is supported by the sorts of selective or fragmentary information and indeterminacies typical of sketches; the abstraction and indeterminacies help in preserving or suggesting alternatives.

A feature which distinguishes informal design representations from more formal capture (such as CAD drawings) is the explicit indication of 'provisionality', that what is being represented is not fixed, not certain, not fully specified, not fully defined, undecided or uncommitted – but is subject to reconsideration and/or alteration. Designers use different qualities of line (e.g., light pressure, broken or wavy lines, different colours), annotations (such as question marks or lists of alternatives or other considerations), and juxtaposition of alternatives as ways of conveying provisionality. The expression of provisionality plays a role in focusing attention (and diverting attention), considering alternatives (including marking things for later consideration of alternatives), and deferring decisions. It allows designers to consider 'downstream' decisions before having all 'upstream' issues resolved.

This ability to defer decisions (and to note them as deferred) is part of designers' dialogue with incomplete ideas – supportive of the creativity of software design by allowing designers to reason their way through parts of the design while setting aside constraints that may impinge from other parts, and supportive also of subsequent systematic exploration and evaluation of the whole.

Fish and Scrivener [11] argued that sketching includes "tolerances and indeterminacies in ways that can amplify the artist's ability to perceive or imagine many options" (p. 117). Something similar appears to apply to software designers, who use representations of 'provisionality' to assist their design exploration and discussion. Information such as crossings out (indicating exclusions, corrections) remain visible and are referred to during discussions. Designers make use of such 'litter', of artefacts remaining from previous activity and discussion, to assist them in recalling the design process, history, and rationale.

4.3 *Insight:* Designers use scenario sketches to make context explicit in the design process.

Design teams often represent context – using scenario sketches, hierarchy and structure diagrams – both as scene-setting and in discussion of specific design decisions, alternating between representations of context and representations of specific design elements. They draw scenarios: use-oriented views of the whole system in context. These show up most frequently in representations generated during discussions between team members. They demonstrate a recurrent attention to context and user needs during design.

5. DELIBERATE PRACTICES TO PROMOTE INNOVATION

In an analysis of effective multi-disciplinary engineering firms, Petre [25] identified 14 'disciplines of innovation', of deliberate practices, whose purpose is to support innovation in design. Most of them are ways to expand the search space, either by admitting more potential solutions, or by broadening the definition of the problem. Some (such as examination of barriers and the systematic relaxation of constraints) are ways to change perspective, to alter the view of the problem or of what might constitute a solution. Some (such as collecting 'loose possibilities') are ways to maintain the knowledge base. Some of these (such as patent searches) are routine and wide-spread practices even in much less innovative companies, but some (such as reasoning about 'essences' or functional abstractions) require exceptional information or expertise.

1. systematic knowledge acquisition (patent searches, technical literature reviews, analysis of legislative requirements and regulatory standards, review of the competition)
2. collection of 'loose possibilities' (keeping track of knowledge, techniques, ideas or opportunities that might come in useful)
3. record keeping (keeping track of design documents of all sorts, meeting notes, informal notes of design rationale, photographs of whiteboards, search and research results, and so on)
4. reflection on completed projects (debriefing on recently completed projects, reviewing potentially relevant past projects, reviews on general themes)
5. systematic re-use or re-implementation of recent innovations
6. identification of barriers (and seeking to remove them in order to identify previously unnoticed assumptions, to review the status of existing limitations on technologies, and so on)
7. attention to conflicts (such as those between stakeholder goals, or between constraints, in order to expose assumptions or seek a new alignment)
8. brainstorming
9. systematic exploration of possibilities (finding gaps, or finding unexplored relationships between problems and potential approaches)
10. scenario-based reasoning to explore assumptions and consequences
11. stripping down to fundamentals (setting aside the 'noise' of detail and stripping the problem down to what essential functionality must be addressed and how it might be achieved)
12. considering 'essences' (or functional abstractions)
13. systematic variation in constraints
14. playing with toys (investigating other people's widgets and developing pet ideas)

It is striking how many of these disciplines require expertise, either in terms of breadth and depth of experience, or in terms of expert reasoning, such as the identification of deep structure in problems and solutions.

In software design, experts use some similar strategies to deal with intractable problems, the sorts of design problems that others find 'too big for one head' [Petre, 26]. Their strategies include:

a. simplification (solving a simpler problem)
a. transformation (into another, easier to handle, form)
b. re-segmentation (dividing it up differently, identifying sub-goals)
c. relaxing constraints (altering or reducing constraints; changing reality)

d. analogy (using an analogy to provide some key insight about functionality, or using differences between apparently analogous thing to explore boundaries)
e. abstraction (solving the essence, for example identifying one key functional abstraction)
f. re-shaping the problem space (solving the problem instead of the solution)
g. seeking insight (rather than solution per se, for example solving a different but related problem, or proving that the problem maps onto a known insoluble problem)

As with the 'disciplines of innovation' these are creative strategies concerned with moving around the design space. Some are (such as relaxing constraints) are about divergence, or expansion of possibilities. Others (such as analogy) are about pruning the space, making progress within a conception of the problem, and converging toward a solution.

Cultivating awareness of alternatives: Innovation and design require divergent thinking, at least in the early stages, and these studies of expert designers provide evidence of such thinking during the software design process. Embodied in the observed expert practices are strategies that help designers to cultivate alternatives or at least maintain awareness that alternatives are possible.

The role of error and failure in innovation: Exceptional companies hire the best possible people and immerse them in a cooperative, communicative culture where exploration is supported. A feature of such cultures is a tolerance for error, supported by robust practices that make it likely that slips and errors will be detected, and a recognition that failure can be informative. Embodied in the 'disciplines of innovation' are reflections on experience that include reflections on failure (as well as successes), not just to learn from them, but also to detect when conditions have changed in a way that might re-open possibilities. One of the advantages of play is that it allows designers to explore ideas before they're needed, and hence with little penalty for failure.

6. DISCUSSION: TENSIONS BETWEEN OBSERVED SOFTWARE DESIGN PRACTICES AND SOFTWARE ENGINEERING METHODOLOGY

Several themes have recurred through these investigations: the role of explicit provisionality in maintaining alternatives and exploring the design space; the role of juxtaposition in exploration of design ideas; the awareness of context and the attention to user needs; the balance of activity and strategies between expansion of the space of possibilities and convergence toward a particular solution; the role of dialogues in design, and the social components of effective performance.

Examinations of professional practice make it clear that professional software designers, and in particular experts, don't necessarily do what the literature would suggest that they do. Yet all of the teams and individuals we studied were reflective practitioners [Schön, 33], who engaged in systematic practices including the examination and evaluation of their own work and working.

6.1 Systematic practice vs. software development methodology

Few of the teams studied consistently followed a specified software development methodology. This is not to say that that they were not methodical – on the contrary, high-performing individuals and teams demonstrate deliberate, systematic practices with due attention to both functional and non-functional requirements, to context, and to the needs of various stakeholders, from users to maintainers.

Moreover, their practices are not orthogonal to 'received' methodologies. On the contrary, they have elements in common. This is no surprise, given that methodologies are for the most part un-startling and un-revolutionary, but rather tend to provide a rationalized, coherent account of something practitioners have already been attempting. So, for example, although the traditional teams studied during this programme of research did not engage in pair programming, many did routinely engage in pair debugging. In another example, most use formal methods, but they do so selectively, and only when the potential benefit warrants the investment.

These software designers showed *no* reluctance to investigate potential tools and methodologies, and they routinely scan the literature and review applications. It must be considered that these are high-performance teams, with well-established methodologies and work practices. They continually seek tools and methods that augment or extend their practice, but they are reluctant to change work practices (particularly work style) without necessity. The tools that persisted and passed into use were those which were robust, worked at scale, and associated well with their preferred software development environment. The take-up of tools and methodologies was based on a thoughtful cost-benefit analysis which took into account the impact on the teams' own systematic practices. Tools for which the cost of take-up was perceived as too high were discarded. Usually the cost was associated with taking on the philosophies, models or methodologies associated with the useful elements. Where there were major discrepancies of process between the package and existing practice, or where there was incompatibility between the package and other tools currently in use, the package was typically discarded as likely to fail. It should be noted that 'Not invented here' was *never* offered as a reason not to use a tool.

6.2 Deliberate changes of paradigm, formalism, and representation

A paradigm is a decision about what to see, a kind of formalism, a focus on particular aspects of a problem. It makes some things more accessible by pushing others back. Hence, no one paradigm will suit every problem; no one paradigm will make easier the whole set of problems that people solve with computers.

Expert programmers use a paradigm as a thought-organizer or a discipline or a frame of reference. They collect a repertoire of useful paradigms — of *reference models* — which offer different views onto which problems can be mapped, and which facilitate different aspects of solution and different virtues.

Petre and Green [30] introduced the need to "escape from formalism" as an essential part of real-life, professional-level design, necessary to cope with things not accessible within a given formalism. Changing paradigm is a mechanism for escape

from one formalism — from one set of constraints or values — into an alternative. Expert programmers employ a conscious change of paradigm in order to re-assess a solution or to gain insight—that is, they exert a paradigm to reveal the information they know they want or to support the reasoning process they know they need to accomplish. For them, a paradigm is used as a convenient (if temporary) world view, a way of looking at things, a way of doing things—a decision about what the world is, for the moment. But the sense is of a pair of lenses rather than of the Kuhnian set of eyes—something one can put on, take off, alternate with something of a different hue; not necessarily something incorporated permanently and to the exclusion of alternatives. The expert treats a programming paradigm as a reasoning tool.

6.3 Reassessing the trajectory

One contrast between expert software designers' deliberate practices and software engineering methodologies, is the process of continual reassessment. Expert behaviour includes significant elements of reflection, correction, and reassessment of the design problem. Software engineering methodologies are largely about setting a solution trajectory and following it through. A colleague has described them as 'a juggernaut'. They tend not to be about creativity, or reflection – methodologies are more concerned with 'normal' design (which after all dominates the field) and with 'normal' designers than with 'radical' design and 'innovative' designers. In contrast, expert practice is highly reflective, and gives due attention both to seeking insight along a suggested trajectory (about making progress within a conception of the problem, and a notion of the destination) and to reconsidering the trajectory as understanding develops. In my colleague's analogy, the expert designers periodically 'get off the juggernaut', examine the landscape, and reassess both trajectory and destination.

One use of scenarios is to encourage designers to change perspective away from a product and to look instead at the whole process in which a product might play a role, in order to identify which and how many steps in the process a new product might cover. This is often characterised as 'white space finding': examining a whole process (perhaps one that is not yet achievable) and trying to identify inefficiencies, obstacles, or gaps in the currently available products or services that limit the process.

6.4 Social context

Another contrast between expert practice and software engineering methodologies is the attention to social knowledge. Methodologies may well define roles in the development process, but they say less about interactions. And yet study of high performing teams makes it clear that the interplay between designers plays a crucial part both in nurturing creativity and innovation and in embedding systematic practice and rigour. High-performing teams use knowledge of the group, of both individual and combined strengths and limitations, to structure their activities and reinforce their strengths. Exceptional teams take care over the deliberate induction of new members into local culture and practice, while eliciting fresh perspectives from them. Care is also taken over deliberate knowledge recovery from exiting members before they leave, although the collaborative, reflective culture tends to ensure that project and process knowledge is disseminated among personnel. Reliance on qualities such as expertise, reliability, and trust, and on practices such as pair debugging that provide systematic checks on activities and sharing of knowledge, can liberate individuals to extend themselves.

6.5 Conclusion

Looking at software design as it is practised by experts and high-performing teams reveals not only a variety of useful strategies but also essential characteristics of the design process. Designers make use of provisionality and juxtaposition to explore alternatives and maintain awareness of options. They deliberately change paradigms, formalisms and representations as a way of changing perspective. And they resist tools that impose too severely on their work practices. Many of their strategies concern expansion of the design space, not just convergence to a solution. More work is needed on supporting conceptual design and providing conceptual design visualizations. Design is a process of dialogues: between designers and artifacts, and among designers. These have implications for tools and methodologies, suggesting that design tools should promote the dialogue between designer and representation, that variations on formalisms might usefully be supported, that explicit provisionality and juxtaposition are essential features, that fluid transitions and mappings between conceptual and software representations are likely to be beneficial, and that capturing and exploiting domain knowledge is a challenge to be addressed.

7. ACKNOWLEDGMENTS

The author is profoundly grateful to the expert software designers and teams, without whom the paper would not be possible, and to their companies which permitted access. The author is a Royal Society Wolfson Research Merit Award Holder. Some of research on which this paper draws was carried out under grants including: EPSRC grant GR/J48689 (Facilitating Communication across Domains of Engineering) in collaboration with George Rzevsky and Helen Sharp, and EPSRC Advanced Research Fellowship AF/98/0597. Thanks also to Bashar Nuseibeh, Andre van der Hoek, David Bowers, Lutz Prechelt, Thomas Green and Jim Buckley.

8. REFERENCES

[1] Allwood, C.M. 1986. Novices on the computer: a review of the literature. International Journal of Man-Machine Studies, 25, 633-658.

[2] Ball, L.J., and Ormerod, T.C. 2000. Putting ethnography to work: The case for a cognitive ethnography of design. International Journal of Human-Computer Studies, 53, 147-168.

[3] Beck, K. 1999. Extreme Programming Explained: Embrace Change. Addison-Wesley.

[4] Beyer, H., and Holtzblatt, K. 1988. Contextual Design: Defining Customer-Centered Systems. Morgan Kaufmann.

[5] Boehm, B.W., 1981. Software Engineering Economics. Prentice-Hall 'Advances in Computing Science and Technology' series. Prentice-Hall. xxvii, 767.

[6] Bucciarelli, LL 1988. An ethnographic perspective on engineering design. Design Studies, 9, 159-168.

[7] Craft, B., and Cairns, P. 2006. Using sketching to aid the collaborative design of information visualization software. In

Human Work Interaction Design: Designing for Human Work, IFIP vol. 221, 103-122.

[8] Cross, N. 2006. Designerly Ways of Knowing. Springer-Verlag.

[9] Curtis, B., Krasner, H., and Iscoe, N. 1988. A field study of the software design process for large teams. Communications of the ACM, 31, 11, 1268-1287.

[10] Damm, C.H., Hansen, K.M., and Thomsen, M. 2000. Tool support for cooperative object-oriented design: gesture based modeling on an electronic whiteboard. In Proceedings of the SIGCHI conference on Human Factors in Computing Systems, ACM, 518-525.

[11] Fish, J. and Scrivener, S. 1990. Amplifying the mind's eye: sketching and visual cognition, Leonardo, 23, 1, 118-126.

[12] Flor, N.V., and Hutchins, E.L. 1991. Analysing distributed cognition in software teams: a case study of team programming during perfective software maintenance. In J. Koenemann-Belliveau, T.G. Moher and S.P. Roberston (eds), Empirical Studies of Programmers: Fourth Workshop, Ablex.

[13] Goel, V. 1995. Sketches of Thought. MIT Press.

[14] Goldschmidt, G. 1991, The dialectics of sketching. Creativity Research Journal, 4, 2, 123-143.

[15] Hamming, R.W. 1987. Numerical Methods for Scientists and Engineers, 2nd ed., Dover Publications.

[16] Kaplan, S., Gruppen, L., Leventhal, L.M., and Board, F. 1986. The Components of Expertise: A Cross-Disciplinary Review. The University of Michigan.

[17] Ko, A.J., DeLine, R., and Venolia. G. 2007. Information needs in collocated software development teams. In 29th International Conference on Software Engineering (ICSE '07), IEEE Computer Society, 344-353.

[18] Lansdown, J. 1993. Visualising design ideas. In Proceedings of Interacting with Images (London, February). BCS.

[19] Lawson, B. 2003. Schemata, gambits and precedent: some factors in design expertise. In N Cross and E Edmonds (eds), Expertise in Design: Design Thinking Research Symposium 6, Creativity and Cognition Studio Press, 37-50.

[20] Logie, R.H. 1989. Characteristics of visual short-term memory. European Journal of Cognitive Psychology, 1, 275-284.

[21] Lubars, M., Potts, C., and Richter, C. 1993. Developing Initial OOA Models. In Proceedings of 15th International Conference on Software Engineering, IEEE Computer Society Press, 255-264.

[22] Luff, P., Heath, C., and Greatbatch, D., 1992. Tasks-in-interaction: paper and screen based documentation in collaborative activity. In Proceedings of CSCW 92, 163-170.

[23] Miller, G.A. 1993. Images and models, similes and metaphors. In A.Ortony (ed.), Metaphor and Thought, 2nd edition. Cambridge University Press, 357-400.

[24] Newell, A., and Simon, H.A. 1990. GPS: A Program that Simulates Human Thought. In Computers and Thought, McGraw-Hill.

[25] Petre, M. 2004. How expert engineering teams use disciplines of innovation. Design Studies, 25, 477–493.

[26] Petre, M. 2007. Expert strategies for dealing with complex and intractable problems. Keynote address, Psychology of Programming Interest Group Workshop.

[27] Petre, M. 2009. Representations for idea capture in software and hardware development. Open University Centre for Research in Computing Technical Report.

[28] Petre, M. In press. Mental imagery and software visualization in high-performance software development teams. To appear in: Journal of Visual Languages and Computing.

[29] Petre, M., and Blackwell, A. 1997. A glimpse of programmers' mental imagery. In S. Wiedenbeck and J. Scholtz (Eds), Empirical Studies of Programmers: Seventh Workshop, ACM Press, 109-123.

[30] Petre, M., and Green, T.R.G. 1990. Where to draw the line with text: some claims by logic designers about graphics in notation. In D. Diaper, D. Gilmore, G. Cockton, and B. Shackel (eds.), Human-Computer Interaction: Interact'90, North-Holland, 463-468.

[31] Rittel, H., and Webber, M. 1973. Dilemmas in a general theory of planning. Policy Sciences, 4, 155-169.

[32] Schön, D. 1988. Design rules, types and worlds. Design Studies, 9, 3, 181-190.

[33] Schön, D.A., 1983. The Reflective Practitioner: How Professionals Think in Action, Basic Books.

[34] Simon, H.A. 1996. The Sciences of the Artificial, 3rd ed., MIT Press.

[35] Vincenti, W.G. 1990. What Engineers Know and How They Know It: Analytical Studies from Aeronautical History. Johns Hopkins University Press.